CS 450

Module R3

Next Week

- R2 is due next Friday
  - Make sure to correct all errors with R1
  - Correct errors in the documentation as well, it will be checked and graded again.
  - Make sure, in your sys_init call, you change the parameter to "MODULE_R2"
  - If you have completed any extra credit, make sure you tell me when I am testing your system so I can test it.
- Late demonstrations will not be accepted.
- If a residual problem from R1 prevents the testing of an R2 command, you will lose points.

Documentation Due with R2

- Manuals
  - Programmer's Manual- add on to what you did for R1
  - User's Manual- add on to what you did for R1, do not include temporary command
  - Temporary Commands Manual- same as user's manual but with only temporary commands
- Code
  - Turn in a complete copy of your source code (all .C and .H files, except MPX_SUPT) with your documentation.

R3: Introduction

- R3 is due along with R4 on Friday, Oct. 25th
- We are now ready to implement the functions that are needed to allow the MPX system to execute processes and switch between them
- You will be writing two interrupt handlers
  - The dispatcher will give the first process on the ready queue control of the CPU, thus allowing it to execute
  - The sys_call handler will handle interrupts generated by MPX programs and call the dispatcher to run the next ready process

Review: Interrupts & Handlers

- An interrupt is a way for software or hardware to indicate to the CPU that some event has occurred.
- When an interrupt is generated, the CPU calls an *interrupt handler* to handle that specific interrupt
- The address of all the interrupt handlers in a system are maintained in *interrupt vectors*. The CPU can find the interrupt vector for a specific interrupt using the interrupt's ID.
- Interrupt handlers:
  - Push context info onto a stack –Turbo C automatically does this for you
    - Interrupts can occur at any time, they are not predictable, therefore the complete system state has to be preserved
  - Execute something
    - Interrupt handlers should do only what is absolutely necessary and contain as little code as possible
  - Pop context information from the stack and restores it – Turbo C automatically does this for you
    - Restoring the system state so whatever what running before the interrupt can continue

Interrupt Keyword

- Interrupt keyword (Turbo C Only)
  - Used to designate interrupt handlers; automatically injects assembly code that saves CPU register values into the functions designated as interrupt handlers
  - Not part of the ANSI C Standard, specific to Turbo C
  - Prototype for interrupt handler
    - void interrupt name()
    - Note there cannot be parameters, and there cannot be a return value
- In Module R3, you will write an interrupt handler of your own. You will then use a support software function to set the address in the interrupt vector of an unused dos interrupt (60h) to point to your handler. Therefore, whenever an MPX process generates interrupt 60h, the CPU will call your interrupt handler.

Generating Interrupts in MPX

- MPX processes will use the "sys_req" function to generate an interrupt. When a process generates an interrupt, it will have a status associated with it. For R3/R4- the statuses are
  - IDLE- means the program is not done terminating, so put it back in the ready, not suspended state and reinsert it into the ready queue
  - EXIT- program wants to terminate- so delete its PCB from memory
  - Other statuses, such as READ and WRITE, will need to be handled in Module R6.

Dos.h functions

- You will need to include dos.h
  - #include <dos.h>
- Construct a far pointer
  - unsinged char * MK_FP(unsigned int SEGMENT, unsigned int OFFSET)
- Get the segment of a far pointer
  - unsigned char FP_SEG(void*)
- Get the offset of a far pointer
  - unsigned char FP_OFF(void*)

R3 Overview

- Two interrupt handlers
  - Dispatcher
    - will be called by the MPX OS
  - Sys_call
    - will be triggered by an interrupt
- Temporary Function
  - Load Processes
- Temporary Commands
  - To call dispatcher
  - To load test processes

Context

- When a process is interrupted, the values in all of the CPU registers must be saved. This information will be placed on the stack by Turbo C, but to access it, you will need to define a C structure- the PCB's context.
- Sample is below- note that you can call the structure whatever you want, but you MUST use the same datatypes and have the registers in the same order!

```
typedef struct context {
   unsigned int BP, DI, SI, DS, ES;
   unsigned int DX, CX, BX, AX;
   unsigned int IP, CS, FLAGS;
} context;
```

Context Switching

- An important part of this Module is context switching. Context switching allows the system to replace a process that is currently running with a different process. In a context switch, you must:
  - Save the context of the running process
  - Replace it with the context of the process you want to run
- Turbo C automatically saves 12 registers for you when you use the interrupt keyword. However, the SS and the SP must also be saved, and Turbo C does not do this automatically.
- In Turbo C, you can directly access the SS and SP using _SS and _SP. You should save them into variables of type unsigned short.
- You can assign new values to the SS and SP using the dos functions mentioned earlier:
  - _SS = FP_SEG(sys_stack)
  - _SP = FP_OFF(sys_stack) + SYS_STACK_SIZE
- When assigning new values to _SS and _SP, you should use intermediate variables.
- When accessing (saving or changing) the SS and SP, you must do so in consecutive, uninterrupted steps.

Parameters

- Parameters cannot be passed to an interrupt handler in the normal way.
- When an MPX process generates an interrupt, sys_req will place certain parameters on the process's stack.
- We need a structure to represent the parameters so that we can access them:

```
typedef struct params{
    int op_code; --operation code, IDLE, EXIT, READ, WRITE, etc
    int device_id; --TERMINAL, COM_PORT
    byte *buf_addr; --buffer address
    int *count_addr; --size of the buffer
} params;
```

- op_code is the main concern of ours for now… it will be either IDLE or EXIT, as described before
- Byte = unsigned char
- We will discuss later how to access these parameters

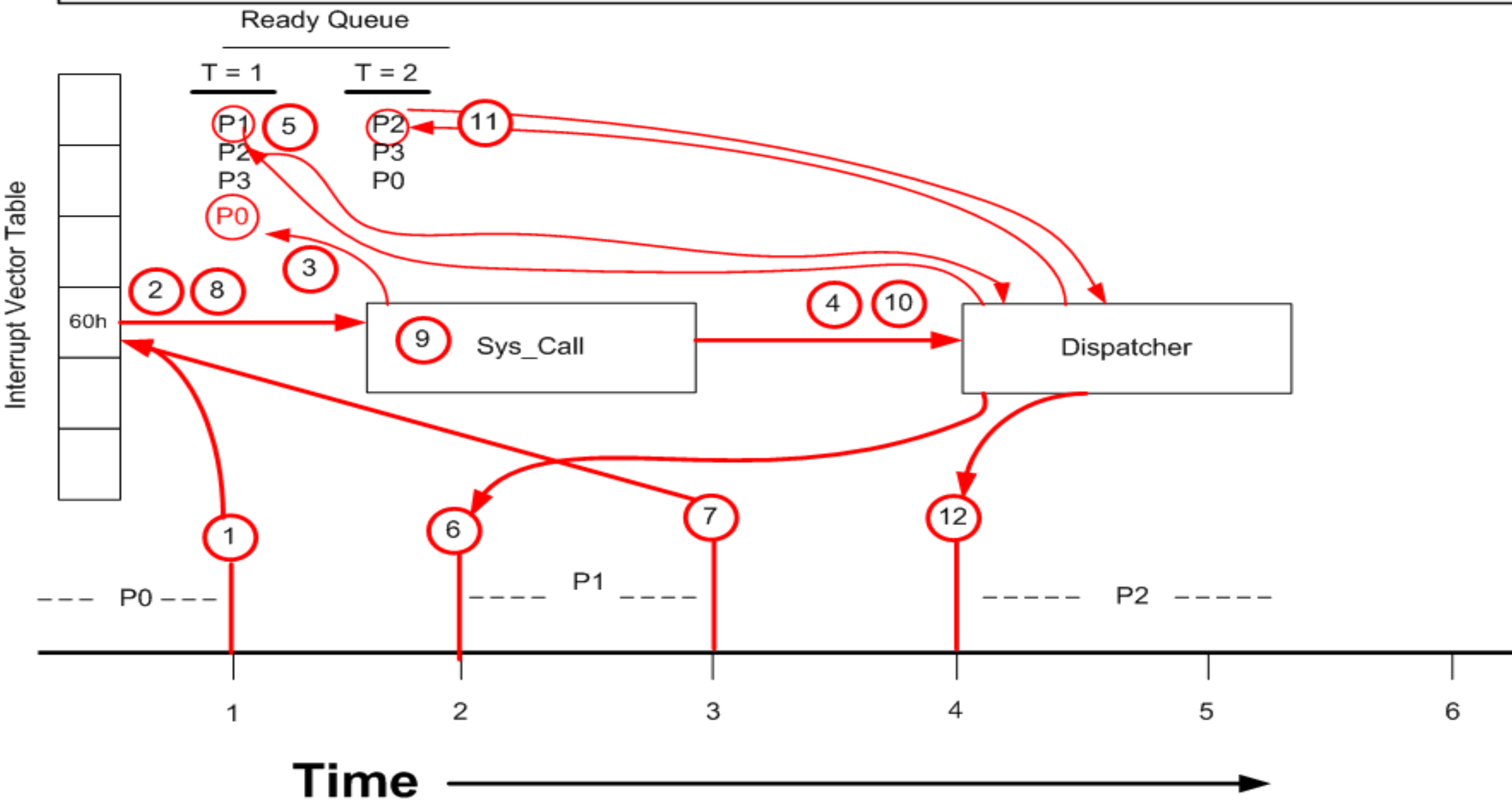
R3: Variables Needed (Example)

- Variables to save and manipulate the stack segment (SS) and stack pointer (SP).
  - unsigned short ss_save;
  - unsigned short sp_save;
  - unsigned short new_ss;
  - unsigned short new_sp;
- The process that is currently running will not be in any queue, but we still need to keep track of it, so we need a variable to hold a pointer to that process's PCB. We designate this "cop" (current operating process)
  - PCB *cop;
- Temporary system stack
  - char sys_stack[SYS_STACK_SIZE];
  - Use a symbolic constant for the sys_stack_size- make it at least 1024

Stack Manipulation using Dos.h

- Save stack pointers
  - ss_save = _SS
  - sp_save = _SP
- Restore stack pointers
  - _SS = ss_save
  - _SP = sp_save
- Switch to temporary stack using intermediate variables
  - new_ss = FP_SEG(&sys_stack);
  - new_sp = FP_OFF(&sys_stack) + SYS_STACK_SIZE;
  - _SS = new_ss;
  - _SP = new_sp;
- Make a far pointer to stack top
  - stack_top = MK_FP(ss_save, sp_save);

1. P0 is executing and generates an interrupt by calling sys_req(IDLE,…)
2. CPU references interrupt vector for interrupt 60h, and calls your sys_call interrupt handler
3. Sys_call looks at the parameters and sees that P0 wants to continue to execute, so it places P0 back on the queue
4. Sys_call calls dispatcher
5. Dispatcher checks the ready queue, sees that P1 is ready to run.
6. Dispatcher performs a context switch, which allows P1 to execute
7. P1 has been executing and generates an interrupt by calling sys_req(EXIT)
8. CPU references interrupt vector for interrupt 60h, and calls your sys_call interrupt handler
9. Sys_call looks at the parameters and sees that P1 wants to terminate. Sys_call frees all memory allocated to P1
10. Sys_call calls dispatcher
11. Dispatcher checks the ready queue, sees that P2 is ready to run
12. Dispatcher performs context switch, allowing P2 to execute…



Procedure: sys_call

- Parameters: none
- Returns: null
- Sample Prototype
  - void interrupt sys_call();
- Sys_call is not directly called. It is invoked (indirectly) by MPX processes requesting to go IDLE or EXIT
- Sys_call interprets system call parameters, puts the currently running process back in a queue or deletes it, depending on the nature of the interrupt.
- Then calls the dispatcher to allow the next process on the ready queue to run.

Sys_call: outline

- When an MPX process generates an interrupt, the sys_req function places the parameters on the stack. In sys_call, you need to access those parameters to uncover the reason the interrupt was generated.
- Access the stack and get the parameters
  - params *param_ptr = (params*)(MK_FP(_SS, _SP) + sizeof(context))
- Switch to temporary stack
- Check the parameters (might be useful to do in a SWITCH statement as we will be adding to this later)
  - If param_ptr->op_code = IDLE
    - Process wants to continue executing
    - Set cop state to ready
    - Insert cop into the ready queue (in priority order)
  - If param_ptr->op_code = EXIT
    - Process wants to terminate
    - Delete cop (free its memory)
    - Set cop = null
- Call the dispatcher

Procedure: Dispatcher

- Parameters: none
- Returns: void
- Sample prototype
  - void interrupt dispatch()
- The dispatcher will be responsible for running processes from the ready queue.
- When the dispatcher is called, this means that the process that is “currently” running doesn’t need to run anymore, and another process should be given control of the CPU.
- The dispatcher is called directly from the sys_call handler.

Dispatcher

- The first time the dispatcher is run, it needs to see if the stack pointers (SS and SP) have been saved yet – if not you must save them so dispatcher has a way of returning to the function which originally called it.
  - Check to see if ss_save is NULL
    - If yes, save the stack pointers (_SS, _SP) to the save variables
- Now, we need to find the next process that will run
  - Find the first process in the ready queue that is not suspended
- If no PCB is ready, we need to restore the MPX state
  - Set the cop variable to NULL
  - Restore the stack pointers (SS, SP) using the save variables
  - Return
- If a PCB is found that is ready, we need to perform a context switch to allow that process to run
  - Set the cop variable equal to the process that is next in line to run
  - Remove the process from the ready queue and set its state to “RUNNING”
  - Context switch
    - Set _SS and _SP equal to the PCB’s stack pointer using dos.h functions
  - Return

R3 Initialization

- We need to set the interrupt vector to point to your system call handler (sys_call). This is so the interrupt instruction generated by MPX processes causes sys_call to be called. If you call your system handler sys_call, you would need to use the command:
  - sys_set_vec(sys_call)
    - Returns 0 if no problem
    - Returns ERR_SUP_INVHAN if invalid handler name
- You should also set all of your stack save variables to null in the initialization.
- You also need to modify your allocate_pcb function (R2), specifically when you set the stack_top pointer, you should set it to
  - Stack_top = Stack_base + Stack_size – sizeof(context)

Test Processes

- On Dr. Mooney's website is a file named "procs-r3.c" This contains 5 test processes we will use to test R3. In order to use these, however, you will need to create a procedure (load_procs) to load these processes into your MPX system.

Load Processes: outline

- Need to do the following sequence 5 times (once for each process). Assumed that you have created a PCB* np, and a context* npc
  - np = Call your setuppcb function, give PCB name (ex “testproc1” or “test1”), priority = 0, class = APPLICATION
  - npc = (context*) np->stack_top;
  - npc->IP = FP_OFF(&test1_R3); //test1_R3 is a func name in procs-r3.c
  - npc->CS = FP_SEG(&test1_R3);
  - npc->FLAGS = 0x200;
  - npc->DS = _DS;
  - npc->ES = _ES;
  - Insert np into the ready queue in priority order
  - Repeat for each test process
- You should also include uniqueness checks on the process names.

Temporary Commands

- Load processes
  - Will call your load_procs() function to load the test processes from procs-r3.c
  - No parameters
  - Suggested commands: load, load_test, loadr3, etc.
- Dispatch
  - Will call dispatcher
  - No parameters
  - Suggest commands: go, dispatch
- Do not combine these commands- you should have one command to load the processes and a separate command to call dispatcher

How to use procs-r3.c

- Put prototypes for the test processes in your R3.C or R3.H file (ex. void test1_R3(void);)
- Link procs-r3.c with your project when you compile, DO NOT INCLUDE IT!
- Use your load command to load the processes.
- Use your dispatch command to run them
- You should see all 5 test processes run with lines like
  - Test n dispatched; loop count = 1
- You should be able to load and dispatch these processes infinitely many times.
- If your system crashes after doing this a couple of times, you more than likely have a memory leak.

Tips and Hints

- Change your “sys_init” call to have the parameter “MODULE_R3”
- Sys call and dispatcher should be short functions. Neither should require more than 30-40 lines of code.
- **Do not use printf statements inside your dispatcher and sys_call functions**. They will cause problems and make your code harder to debug.
- Remember to always manipulate _SS and _SP in consecutive, simple statements.
- Suspended processes and blocked processes should never be run. To test your program, try loading the processes and then blocking/suspending a few before calling dispatch. Only those in the ready, not suspended state should run, and all others should remain in their appropriate queues until they have been run.
- Do not declare variables inside of interrupt handlers- make any variables you need to use in an interrupt handler global.
- Most R3 errors can be traced back to R2 functions
- Read the project manual for R3- there is more pseudocode available in the manual.